Scalable Self-Stabilization via Composition

William Leal Anish Arora

Department of Computer and Information Science[*]
The Ohio State University, Columbus, Ohio, USA
{leal, arora}@cis.ohio-state.edu

Technical Report OSU-CISRC-7/03-TR46

August, 2003
Revised January, 2004

Abstract

Objections to the practical use of stabilization have centered around problems of scale. Because of potential interferences between actions, global reasoning over the entire system is in general necessary. The complexity of this task increases dramatically as systems grow in size. Alternatives to dealing with this complexity focus on reset and composition. For reset, the problem is that any fault, no matter how minor, will cause a complete system reset with potentially significant lack of availability. For existing compositional alternatives, including compositional reset, severe restrictions on candidate systems are imposed.

To address these issues, we give a framework for composition in which global reasoning and detailed system knowledge are not necessary, and which apply to a significantly wider range of systems than has hitherto been possible. In this framework, we explicitly identify

[*] This work was supported in part by DARPA contract OSU-RF #F33615-01-C-1901, NSF grant NSF-CCR-9972368, and a grant from Microsoft Research.

for each component which other components it can corrupt. Additionally, the correction of one component often depends on the prior correction of one or more other components, constraining the order in which correction can take place. Given appropriate component stabilizers such as detectors and correctors, we offer several ways to coordinate system correction, depending on what is actually known about the corruption and correction relations.

By reducing the design of and reasoning about stabilization to local activities involving each component and the neighbors with which it interacts, the framework is scalable. Reset is generally avoided by using the correction relation to check and correct only where necessary. By including both correction and corruption relations, the framework subsumes and extends other compositional approaches.

Though not directly a part of this work, we mention tools and techniques that can be used to help calculate the dependency and corruption relations and to help create create the necessary stabilizers.

To illustrate the theory, we show how this framework has been applied in our work in sensor networks.

1 Introduction

Faults are a fact of life in computer systems. These can include environmental faults such as temperature variations that affect components, server failures, bad input and operator error, components that are mismatched, software bugs, and others. If the system is large, and especially if it is distributed, it is often impractical or impossible to mask the effect of these faults. Yet even if the system misbehaves visibly, we may nevertheless consider it useful if the bad behavior does not continue indefinitely. For example, a wireless network may experience a variety of environmental faults that affect the quality of service delivered to a user, but it is useful if eventually adequate service can be provided.

One way to address the problem is via self-stabilization, where a system always recovers to good behavior from arbitrary faults, provided the faults stop (or at least, stop long enough). Self-stabilization (or just "stabilization") is not new. However, there are objections to the practical use of stabilization, centering around problems of scale. Because of potential interferences between actions during recovery from bad states, global reasoning over the entire system is in general necessary. The complexity of this task increases dramatically as systems grow in size, since a reachability analysis must be performed on the state space, and each action must be checked against every bad state (to ensure convergence to good states), as well as against every good state (to ensure that the system does not leave good states).

Alternatives to dealing with this complexity focus on reset and composition. Reset, in which the system is restored to some pre-determined globally good state, is usually easy to add to a system, but tends to be inefficient at run time since every fault, even minor ones, will cause a complete system reset, with concomitant lack of availability.

Several previous attempts have been made to find a stabilizing compositional framework, but these impose severe restrictions on candidate systems. Some of the main compositional approaches are mentioned next. [20] requires that components be independent and non-interfering, so each component stabilizes independently. [8] permits the correction of one component to depend on another, but component invariants are trivially *true*. Stabilization by layers [13, 17] requires that all components in a hierarchy reach a fixed point except for the top one. Recursive restart [11] is a compositional reset approach that blocks faulty components and those they call until they are all reset; parents of restartable components must be able to tolerate a reset of their children, effectively losing shared history. None of these approaches handle systems with corruption interference, in which in which an uncorrected component can potentially corrupt one that has been corrected. We address the issue by identifying, for the first time, the corruption relation: a binary relation over components that indicates whether a component that is bad can corrupt a good component with which it interacts. We combine this with a correction relation, reflected in some of the existing approaches, that indicates the order in which correction must occur. An additional innovation is that we include meaningful invariants for components as well as link invariants between interacting components; previous approaches have included one or the other, but not both.

Spread of Corruption, and Correction Dependency. The main issue in compositional stabilization is handling cycles of corruption. Fig. 1 shows a system of two linked components, each of which is individually stabilizing. Consider a state where A is not yet stabilized but B is. When they interact, B receives bad input from A and its state becomes bad; we say that *A has corrupted B*. Next, A is corrected to a good state, but since the state of B is bad, it recorrupts A. Next B is corrected, and the system is back to the original state. This cycle can continue indefinitely.

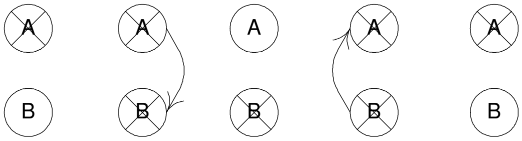


Figure 1: Cycles of Corruption

The corruption cycle in this example could be managed by blocking one of the components, say A, from interacting with B until both had stabilized. Now, suppose that B can corrupt A but not the reverse. Then there is no corruption cycle and blocking is not necessary: all we have to do is wait until B has corrected and then A can correct, bringing the system to a good state. But in some cases there is a correction dependency, as follows. In good system states, there is a link condition between A and B that must hold and hence must be established for the system to be correct. Usually this condition cannot be established in an arbitrary order, but one component, say A, must first become good, and then B can correct itself in relation to A. In this case, B depends on A. The need to establish this link condition while observing the *correction dependency* can lead to corruption cycles. In the following example, a mobile pursuer attempts to reach a destination by traversing a network of tracking sensors that cooperate to give the pursuer a shortest path to the destination.

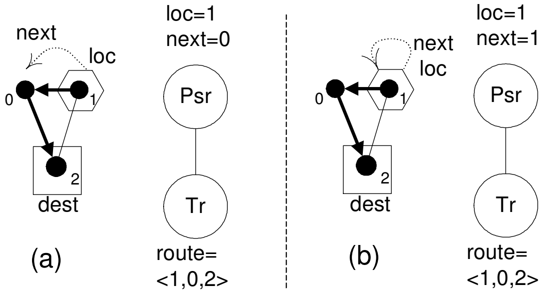


Figure 2: Cycles due to Correction Condition

Pursuer Example. There is a pursuer that must reach a certain fixed destination over a network of nodes. To do this it relies on a tracker to determine the next step. The task of the tracker is to provide a shortest-path route from the pursuer to the destination,

> taking into account the next step that the pursuer has stored. If the tracker finds that the route is invalid, it forms a new one, beginning with the pursuer's location and next step.

In Fig. 2 we have two states. In the first state (a), the destination is at node 2 and the pursuer is at node 1 with the next step to node 0. The components are given on the right, with a matching node diagram on the left. The tracker's route is $\langle 1, 0, 2 \rangle$. Both pursuer and tracker are in bad states. In this state, the tracker attempts to correct the route by forming a shortest path from the next step. However, the resulting route is no different than the current one, so there is no change of state. Next the pursuer sees that it is in a bad state and sets the next step equal to the current location (b). If at this point the tracker corrector were to run, a valid route would be formed. However, before this happens, the pursuer reads the route information and sets the next step to 0, so we are back at state (a). This cycle can continue indefinitely.

We can break the cycle by using blocking. When there is a problem, the tracker blocks by refusing to give the pursuer a next step. After the pursuer has corrected itself, the tracker can form a valid route and then unblock.

The cycle that is introduced in this example comes because the correction dependency (from pursuer to tracker) is opposite the flow of corruption (from tracker to pursuer). If the two relations are in the same direction, then no cycle forms and there is no need for blocking, as the following example illustrates.

> *Producer-Consumer* We have a producer component that creates data for a consumer component. If the producer behaves badly it could create bad values that can make the consumer behave badly.

The consumer depends on the producer since correction of the latter depends on the former being correct. The producer can corrupt the consumer but not the reverse. If both components are stabilizing, then the consumer can stabilize after the producer stabilizes, and no blocking is necessary. Note that this differs from stabilization by layers [13, 17] since the queue does not reach a fixed point.

Methodology. Since our focus is on composition, our methodology assumes that appropriate component stabilizers such as detectors and correctors have already been developed. We offer several stabilization coordinators that can be composed with a system's components to effect system stabilization; the choice of coordinator depends on what is known about the correction and the corruption relations. In general, we assume that correction dependencies form an acyclic structure (a tree, or more generally, a dag). Many systems have this structure; when dependency cycles do exist, they can sometimes be made acyclic [8]. Based on what we know about how correction spreads, we apply an appropriate coordinator to each component. A coordinator consists of actions that are added to each component to control the detection and correction, using blocking where necessary. We give three coordinators: one that avoids blocking altogether, another that uses blocking on a local basis only, and a third that initiates a global correction wave.

Coming up with the detectors, correctors and relations can be complex, but it is a task that is local to a component and its neighbors, rather than being global to the entire system. Hence the development and verification efforts are scalable.

The run-time performance of the coordinators during system stabilization is no worse than that of existing approaches. Likewise, the run-time impact on a system that is in a good state is no worse than that of existing approaches, since detectors can be scheduled to execute at whatever frequency is desired.

We have developed this theory in part as a result of our work in sensor networks [12, 5], which we draw on for examples in the paper.

Summary. Our compositional approach has the following features. It is local, since only knowledge of a component and its neighbors is required. Assuming the necessary stabilizers are available, it requires only limited knowledge of how components interact, and no knowledge of component source code or other internals. It is efficient when there is no problem in the system since detection can be run at any desired rate in any part of the system: more aggressively in areas known to be error-prone, less aggressively elsewhere. It maximizes system availability by unblocking components as soon as it is safe to do so.

Contributions. Our main contribution is that we use composition to eliminate an important barrier to the practical use of stabilization. The technique is local to each component, so the complexity of the development and verification tasks are linear with respect to the number of components in a system.

We accomplish the composition by identifying, for the first time, the corruption relation between components. Based on the dependency and corruption relations, we provide a library of coordinators that manage the system stabilization, using blocking as necessary with varying degrees of efficiency. These give us ways to achieve efficient stabilization without detailed system knowledge.

In addition, we bring together for the first time the ideas of component and link invariants, thus extending the range of systems that can be considered.

Road Map. The rest of the paper is organized as follows. We describe the model of computation in Section 2 and present our approach to composition in Section 3. In Section 4 we discuss how the compositions can be applied in conventional programming settings. In Section 5 we give a case study, drawn from our work with sensor networks. In Sections 6 and 7 we discuss related work, and conclusions and future plans, respectively. Extended discussion and the correctness proofs are in the Appendix.

2 Model of Computation

Next we give the computational model. We assume interleaving semantics with method calls for interaction. We use method calls in order to simplify the presentation and avoid needless complexity, but it is not fundamental to the theory. As we discuss further in Section 4, other communication approaches such as shared memory, communication registers, or message passing could be used instead.

A state space Σ is defined in the usual way, with variables over domains. State predicates are first-order formulae over the variables in Σ. A predicate p is *satisfied* in a state s, written $s \models p$, if the predicate evaluates to *true* in that state; this is extended in the natural way to sets of states.

For communication we use method calls. When a component calls a method on another component, it executes atomically. Methods have interfaces of the usual sort, and are divided into functions that return values and procedures that do not.

Definition 1 (Component) *A component Q for state space Σ consists of the following. (1) A set of* local variables*, $V.Q$, each of which is a variable of state space Σ. (2) A set of* actions*, $A.Q$, given as guarded commands that can read and write the variables of Q and can invoke methods on other components. The* local actions *of Q are those that do not invoke methods.*

(3) A set of methods*, $D.Q$, divided into* procedures *and* functions*. Each method has an interface and an associated command. (4) A state predicate, $I.Q$, called the* component invariant*, that is defined over the local variables of Q and the environment variables. $I.Q$ is preserved by the local actions of Q, where preservation is defined below.*

The state space induced by the local variables over Σ is the *local state space* of Q. Two components are *compatible* if they have the same state space, disjoint local variables, and disjoint methods. Note that any two components can be made compatible by renaming their local variables and methods if necessary, and by extending the state space of each to include the variables of the other.

There is a *link* between components Q and R if an action of either component calls a method of the other; we say that Q and R are *neighbors.* Given a set of compatible components, we let K stand for the symmetric binary *link relation* designating the links between the components, with $K.Q.R$ standing for $(Q, R) \in K$. For each link $K.Q.R$ there is a *link invariant* $T.Q.R$, a state predicate that references at most the local variables of the linked components and the environment. Notationally, $T.Q.R$ is the same as $T.R.Q$.

We say that a component Q is *good* in a state if its invariant is satisfied and is *bad* otherwise, and similarly for link invariants. An *environment* Z is a special component that can read and write any system variable directly.

Definition 2 (System) *A system S consists of the following. (1) A set of pairwise-compatible components, $\mathbb{Q}$, that includes an environment, and is over state space Σ. (2) A set of link invariants $\mathbb{T}$.*

Definition 3 (Composition of Components) *Given components Q and R in a system S, the composition of Q with R is defined as the component with the same state space as both components, whose variables are the variables of both components, whose actions are the actions of both components, and whose component invariant is the conjunction of the components' invariants $I.Q$ and $I.R$ with the link invariant (if any) between them. Composition is commutative and associative.*

The links of a composite component are those links of constituent components to components outside the composite. In Section 3.4 we use this idea to improve the efficiency of stabilization.

An action over a state space may be interpreted as the set of state pairs (s', s) that satisfy the precondition and postcondition of the action. This gives us the *transitions* of a component. The *computations* of a system consist of the set of maximal sequences $\langle s_0, s_1, \dots \rangle$ s.t. for all i, (s_{i-1}, s_i) is a transition of a constituent component of the system.

A predicate p is *preserved* by a transition (s', s) if $(s' \models p) \Rightarrow (s \models p)$. It is preserved by an action of a component if it is preserved by all transitions of the action. It is *closed* in a system if it is preserved by all component actions.

We define the system invariant shortly. We want this to be a closed predicate: in every computation, once the invariant holds, it should hold continuously thereafter. To accomplish this, we require subsystem preservation, where a subsystem is a component along with all the components to which it is linked.

Definition 4 (Subsystem Preservation) *A system preserves its subsystems if the following holds for each pair of linked components Q and R. From any state s' s.t. $s' \models I.Q \ \wedge \ T.Q.R \ \wedge \ I.R$, the actions of Q and R preserve $I.Q$ and $I.R$, as well as any link invariants incident to Q or R.*

Theorem 5 (System Invariant) *Given a system S, let $I.S$, the system invariant, be the conjunction of the component and link invariants of constituent components. Then S preserves its subsystems if and only if $I.S$ is closed.*

Proof. First, assume $I.S$ is closed. Then subsystem preservation follows immediately. Second, assume subsystem preservation applies. Consider a transition (s', s) of some system computation where $s' \models I.S$. Suppose, for contradiction, that $s \not\models I.S$. Then $s \models \neg I.S$, so in s, some component or link invariant is violated. But this contradicts the assumption of subsystem preservation. □

This setup implies local detection and correction [10]. Since the system invariant is given as the conjunction of component and link invariants, violation of the system invariant means violation of a component or link invariant, and correcting the system means correcting component and link invariants. Not all systems have this property. In Section 3.4 we discuss how to address the situation where some parts of the system do not have the property.

3 Component Coordinators for System Stabilization

Our methodology for adding stabilization to a componentized system is to add coordinator actions to each component so that the overall result is a stabilizing system. The coordinators depend on the correction dependencies between components and on the ways components can corrupt each other. The coordinator actions call on component stabilizers for detection, correction, reset, and blocking. We begin by making these ideas precise.

3.1 Component Relations and Stabilizers

Definition 6 (Component Dependency) *Let P and Q be linked components in a system. Let u be a local state of P s.t. $u \models I.P$. If there is a local state v of Q s.t. $v \models I.Q$ and $(u \cup v) \models T.Q.P$ then Q* depends on P for correction*.*

We define the *correction dependency relation*, L, as the set of all (P, Q) s.t. Q depends on P for correction, where we call P the *parent* and Q the *child*; $L.P.Q$ stands for $(P, Q) \in L$. The correction dependency relation (or just "dependency relation") gives us the order in which correction must take place. Unlike the K relation, the L relation need not be symmetric; it it is, then the components are *independent*, which will be the case whenever the link invariants are each *true*.

Definition 7 (Flow of Corruption) *Let P and Q be linked components in a system. Let s' be a system state s.t. $s' \models (\neg I.P \vee \neg T.P.Q)$. Then P* can corrupt Q *if an action of P or Q that invokes a method the other results in a state s s.t. one or both of the following hold. $s' \models I.Q$ but $s \models \neg I.Q$, or for some neighbor R of Q, $s' \models T.Q.R$ but $s \models \neg T.Q.R$.*

We define the *corruption relation* M as the set of all (P, Q) s.t. P can corrupt Q. Informally, if P corrupts Q then it makes Q worse in some way.

Now we define the stabilizers. These are programs that can be called by the coordinator actions to stabilize the system. Different coordinators use different stabilizers.

- Component detector, $cd.Q$. This is a function that returns *true* iff Q's invariant is violated in the current state.

- Link detector, $kd.Q.P$. Assume that P is a parent of Q in L. $kd.Q.P$ is a function that returns *true* iff the link invariant $T.Q.P$ is violated in the current state.

- Local corrector, $lc.Q$. This is a program that establishes $I.Q$. If Q has a parent P in L, and if $I.P$ holds when the corrector is started, then it also establishes the link invariant $T.Q.P$.

- Reset, $r.Q$. This is a program that sets the component to an initial state.

- Blocking, $b.Q$. This is a procedure that can be called by Q or any parent of Q. Running $b.Q(1)$ blocks Q, while $b.Q(0)$ unblocks it. When a component is blocked, its methods are blocked so neighbors cannot access the component state. Overloading notation, $b.Q$ can be used as a Boolean function indicating whether Q is blocked or not.

For convenience we define the *local detector*, $ld.Q$, as a composite detector made up of the disjunction of the component detector and all the link detectors for parents of Q. It detects whether there is a problem with the component or any link: $\neg I.Q \vee (\exists P : L.P.Q : \neg T.P.Q)$.

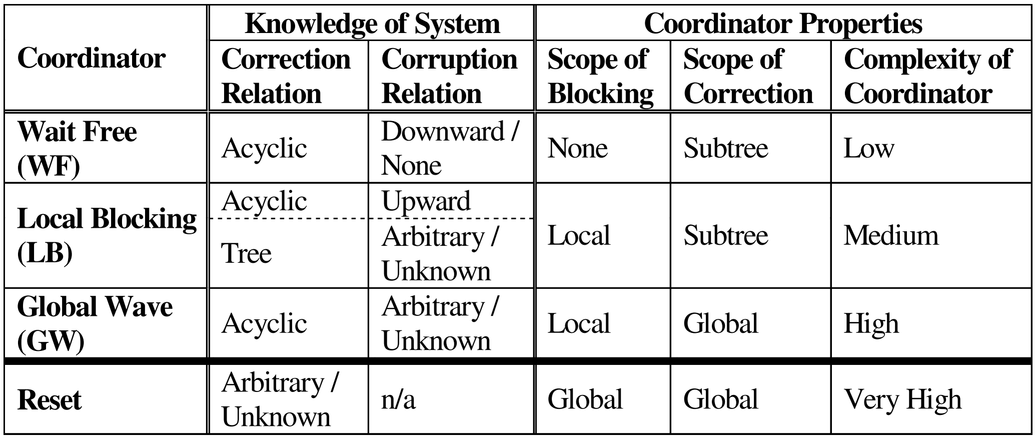
3.2 Stabilization Methodology

Our approach to adding stabilization to a componentized system is as follows. First, we gather information about the system. Next, we check whether we can use one of the available coordinators. If so, then the coordinator actions are customized for each component and added to the component. The result is a stabilizing system.

Gather Information. We want to use the most efficient coordinator that is applicable, so our first task is to identify the system's dependency and corruption relations.

If we don't know these relations, we might be able to calculate them. Given the source code for a component's correctors, we can determine the dependency relation by checking which other components are called. Determining the corruption relation typically requires a more semantic analysis. Assuming we know the local and link invariants, we can check the actions for component P and those of its neighbors in states where P's invariant is violated. If from those states, an action of Q that invokes a method of P

results in the violation of $I.Q$ or $T.P.Q$ then there is corruption from P to Q. This calculation is local and does not require global reasoning. Tools such as Behave! [2] and Magellan MaX [4] can be used to help determine the relations. Approaches such as [14] can be used to help calculate invariants.



Coordinator	Knowledge of System		Coordinator Properties		
	Correction Relation	Corruption Relation	Scope of Blocking	Scope of Correction	Complexity of Coordinator
Wait Free (WF)	Acyclic	Downward / None	None	Subtree	Low
Local Blocking (LB)	Acyclic	Upward	Local	Subtree	Medium
	Tree	Arbitrary / Unknown			
Global Wave (GW)	Acyclic	Arbitrary / Unknown	Local	Global	High
Reset	Arbitrary / Unknown	n/a	Global	Global	Very High

Figure 3: Basic Coordinators

Select Coordinator. Based on the dependency and corruption relations, we choose a coordinator from Fig. 3 that will be applied to each component. If we choose one of the first three coordinators (WF, LB, GW), we must have the correctors and detectors mentioned above, for each component. If we choose LB or GW, then we must also have blocking available for each component. If we choose Reset, then we must have the reset and blocking stabilizers available. If these conditions do not hold then our framework cannot be used to add stabilization to the system.

If a necessary stabilizer does not exist, it might be possible to create it, an activity that is outside the scope of this paper. In Section 7 we discuss refinement of component specifications as a possible way to reduce the complexity of creating stabilizers.

The most efficient coordinator is Wait Free (WF) since there is no blocking. The next best choice is Local Blocking (LB), the coordinator for the pursuer example above, where local blocking is used. The last choice for acyclic dependency relations is Global Wave (GW), where faults initiate a global correction wave is propagated through the entire system. For both LB and GW, blocking is released as soon as it is safe to do so. Note that the correction dependency and the pattern of corruption need not be known in order to achieve system stabilization. However, when they are known, they open the way to potentially greater efficiency.

If we chose the Reset coordinator then we resort to resetting the system, using, for example, the distributed reset of [7]. Any fault, no matter how trivial, will cause a global system reset, with all processing blocked until it is complete. A more nuanced approach to reset is given by [11], where only affected subsystems are reset, but this can still entail substantial system down-time.

Add Coordinator Actions. Each coordinator comes with actions that manage the stabilization process. Once a coordinator is chosen, the actions are customized to the individual component and added into the system.

3.3 Basic Coordinators for Acyclic Correction

Next we give the details for the basic stabilization coordinators that can be applied to systems with acyclic dependency relations. In each case we give the actions that are added to each component, depending on the coordinator.

The proofs that these coordinator actions yield stabilizing systems are given in the appendix. Now, suppose after we have followed the methodology and applied the coordinators that the resulting system in fact does not stabilize. We know the coordinators are correct since their soundness has been proven. So it must be that one or more of the stabilizers is incorrect. [9] addresses this issue with a framework by which a faulty component can be suspected if neighbors discover that it behaves badly even after being corrected.

The coordinators as presented assume that detection, correction and blocking happen atomically. This lets us avoid unnecessary detail in the presentation, and focus on the way the coordination takes place. High atomicity is not central to the theory, however, and lower atomicity implementations are possible. We discuss this in Section 4. We assume strong fairness for detection: component and link detectors will eventually detect invariant violation that occurs sufficiently often.

3.3.1 Wait Free (WF) Coordinator

If the dependency relation is acyclic and the corruption relation is downward, or is empty, then add the following action to each component in the system.

This coordinator is similar to that given in [10].

WF Coordinator

```
$ld$.Q $\longrightarrow lc$.Q                                  %(Detect & Correct)
```

The action given checks whether Q or a link to a parent of Q is violated. In that case, Q and its parent links are corrected. Since corruption is at most downward, there is no need for blocking. An illustration of how the coordination proceeds is given in Fig 4. We mentioned earlier the producer-consumer as an example of downward corruption.

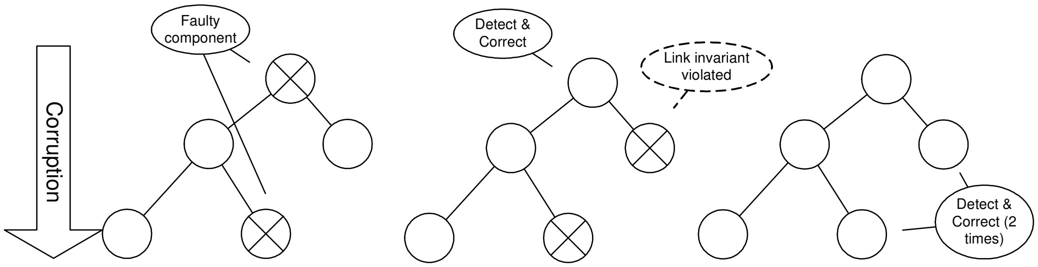


Figure 4: Coordinating Downward Corruption

3.3.2 Local Blocking (LB) Coordinator

If the dependency relation is acyclic and the corruption relation is upward, or if the dependency relation is a tree (or a forest) and the corruption relation is arbitrary or unknown, then add the actions below to each component in the system.

LB Coordinator

```
    $\neg b.Q \ \wedge \ ld.Q \longrightarrow b.Q(1)$                % (Detect & Block)
 []
    $b.Q \longrightarrow$                                     % (Correct & Release)
      (forall $R$ : $L.Q.R$ : $b.R(1)$) ; $lc.Q$ ; $b.Q(0)$
```

If the detector reports a fault, the component is blocked by the first action, which keeps it from corrupting its parents, and marks it as needing correction. If a component is marked for correction, then the second action (a) blocks the children to prevent recorruption of the component, and marks them as needing correction; (b) corrects the component; and (c) releases the block from the component. This not only releases the component to provide service, it protects the component from recorruption and moves the correction process down a level in the dependency relation.

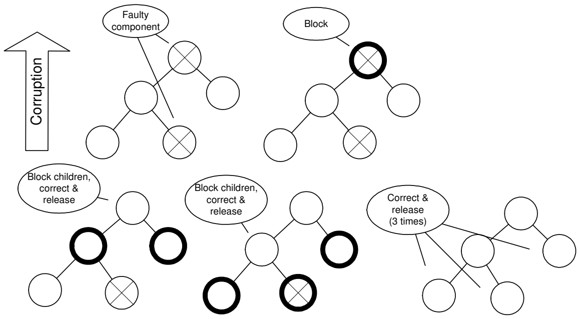

Faulty component
Block
Corruption
Block children, correct & release
Block children, correct & release
Correct & release (3 times)

Figure 5: Coordinating Upward Corruption

Fig. 5 illustrates how the stabilization proceeds when we have upward corruption. Note how blocking forms a barrier to protect the corrected components from recorruption by those not yet corrected. The blocking is local (only boundary components need be blocked) and is quickly released. Hence corrected components are released to resume normal processing as soon as they enter the protected area.

3.3.3 Global Wave (GW) Coordinator

If the dependency relation is acyclic and the corruption relation is arbitrary, or is unknown, then add the actions below to each component in the system. As a reminder, $K.P.Q$ holds if P and Q are neighbors.

We keep two integer variables for each component, *req* to indicate that a correction wave has been requested, and *cor* to indicate the wave has been completed.

Normally all values of *req* and *cor* are the same. When a fault is detected at an unblocked component by the first action, the *req* value is incremented. This has three effects. First, the component is blocked from corrupting its parents. Second, it is marked for correction. Third, it signals a new system-wide correction wave.

After the first action has been executed for component Q, the component will have a *req* value that is higher than any other in the system. The second action causes this new higher value to be spread throughout the system. Eventually it will reach the roots of the dependency relation.

GW Coordinator

```
    $ld.Q \ \wedge \ req.Q = cor.Q \longrightarrow$                 % (Detect & Block)
      $b.Q(1); req.Q := req.Q + 1$
 $[]$
    $K.P.Q \ \wedge \ req.P > req.Q$                   % (Propagate Request)
    $\longrightarrow$
      $b.Q(1); req.Q := req.P$
 $[]$
    $(req.Q > cor.Q) \ \wedge$                          % (Correct & Release)
    $(\forall P : L.P.Q : req.P = cor.P = req.Q) \ \wedge$
    $(\forall R : L.Q.R : req.Q = req.R)$
    $\longrightarrow$
      $lc.Q; b.Q(0); cor.Q := req.Q$
 $[]$
    $(req.Q < cor.Q) \vee$                          % (Fix Inconsistencies)
    $((req.Q > cor.Q) \wedge \neg b.Q) \vee$
    $((req.Q = cor.Q) \wedge b.Q)$
    $\longrightarrow$
      $b.Q(0); req.Q := cor.Q$
```

The third action is the heart of the coordinator. When a new *req* value arrives at a root, it begins a new correction wave downwards. Once the children are blocked, the root is corrected and is unblocked. This proceeds downwards through the dependency relation. If a component Q has multiple parents, $ld.Q$ forces the action to wait until all parents have participated in the same correction wave.

We require that at each component Q, $req.Q$ be greater than or equal to $cor.Q$ and that blocking be consistent with these variables. If a fault affecting the coordinator variables has violated this, then the last action corrects it.

An illustration of how this proceeds is given in Fig. 6. This coordinator is similar to Reset but differs in that the system is not restored to a pre-determined global state. Accumulated history represented in component state can frequently be preserved or, using the correction relation, rebuilt.

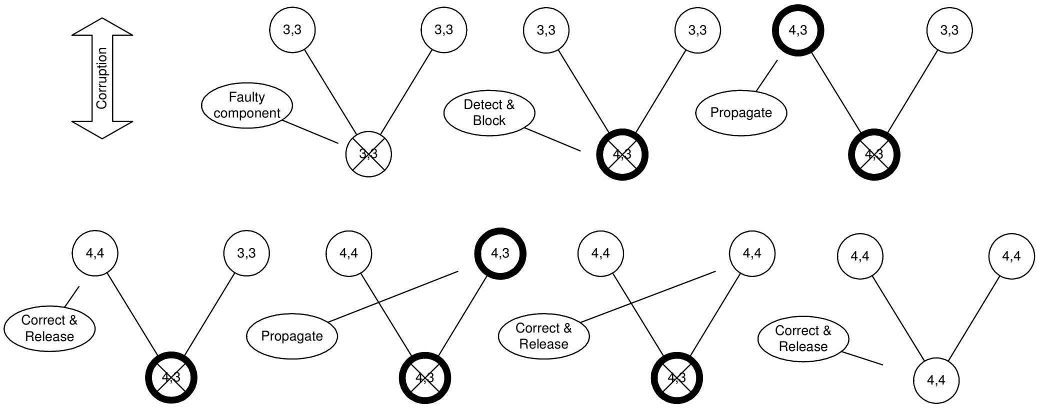


Figure 6: Coordinating Arbitrary Corruption

3.4 Additional Notes

Pass-Through Corruption and Correciton. Consider a system with components A and B joined by a channel component C. The invariant of C is that values are sent in the order they were received. A can corrupt B by sending a bad value via C, and B can be corrected when A is correct and has sent good values to C. However, there is no link between A and B since neither one calls the other directly, and the invariant of C is not sensitive to whether the values it contains are good or bad. Hence we do not have a corruption or correction relation even though corruption and correction in fact happen.

To handle this, we can conjoin the invariant of C with an additional predicate that is true when all the values in C are good values of A. With this stronger invariant for C, we can now have corruption from A to C and from C to B, and similarly for correction.

Coordinating Independent Components. In a system with independent components, the correction relation L is symmetric, so either component can correct itself first. We can project the dependency relation to form a tree (or a forest) and then apply one of the basic coordinators. Since any node can serve as a root in this kind of system, we can try to form the tree so as to minimize blocking. For example, if a system consists of independent components A and B with corruption from B to A, then we could choose A as the root and use the LB coordinator, or we could choose B as the root and use the WF coordinator. Since the latter is more efficient, it would be preferable.

An alternative approach is given by [9]. Rather than form an acyclic structure, components are corrected on a peer basis. If a component detects a fault, it initiates a global recorrection wave, blocking interactions from neighbors that have not yet recorrected. It is similar to the GW coordinator, but the coordinator actions are more complex, and the components must be independent. This approach permits a dynamic relationship between components, in which any component can at any time interact with any other. It contrasts with our approach in which the interactions between components are fixed in advance.

Limiting Spread of Corruption. To maximize system availability we want to limit the spread of corruption through a system. One way to do this is to schedule the detectors for components as we gain experience in a particular system as to which components fail most frequently and which are most reliable. We can be aggressive in checking the former, more relaxed with the latter.

In addition, components with high connectivity can potentially spread corruption more rapidly than others that have fewer neighbors. These components can receive more aggressive attention.

Finally, the structure of the system itself plays a role. In the case of downward corruption, components nearer the roots have more descendants and hence more opportunity to spread corruption. Conversely, with upward corruption, components nearer the leaves have more ancestors and hence more opportunity to spread corruption. In these cases, checking can be more or less aggressive, depending on where the component fits in the dependency structure.

Nested Coordination As defined in Section 2, a subset of the components in a system can be composed into a component. The new composite component is treated as an ordinary component with respect to the rest of the system. We can use this to improve the efficiency of stabilization for some systems, as illustrated by the following extension to the pursuer example; see Fig. 7.

> *Pursuer-Evader Example.* We add an evader to the system. The object of the pursuer is to capture the evader; the object of the evader is to elude capture. The tracking system is in place to assist the pursuer, so the evader is under no obligation to cooperate with it. Hence the evader can move autonomously; when

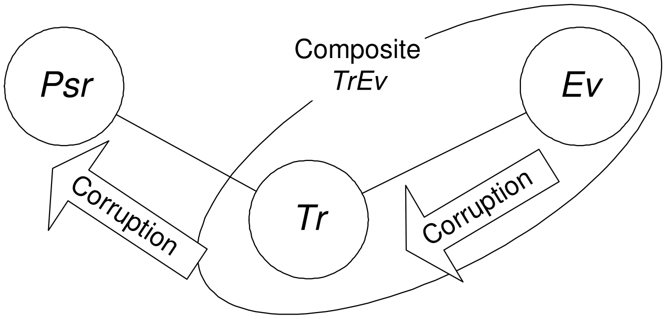


Figure 7: Pursuer-Evader

> it does so, the tracking system's path will no longer be rooted at the evader's location and so it must restabilize.
>
> We add an evader component to the system, *Ev*, that senses the evader's location. This is communicated to *Tr*, which forms a path from *Psr* to the evader's detected location.

If *Ev* falsely indicates that the evader is in one location when in fact it is somewhere else, the path that *Tr* forms will not be a shortest path to the evader and so the link invariant will not hold.

We have corruption from *Ev* to *Tr*, and from *Tr* to *Psr*. Using only the basic coordinators we would have to use GW since the corruption relation is neither upward nor downward. However, we can compose *Ev* and *Tr* into a composite component, *TrEv*, as shown in the figure, to obtain better efficiency. We use the LB coordinator between *Psr* and *TrEv*. Within *TrEv* we use the WF coordinator. The stabilizers for the composite component can be constructed from the stabilizers for *Ev* and *Tr*. For example, component detector $cd.TrEv$ is the disjunction of $cd.EL$, $kd.EL.TS$ and $cd.TS$.

Cyclic Correction. Sometimes cyclic corruption can be handled with the techniques in [8]. If not, nesting can be used along with reset. Figure 8 shows a system with cyclic correction. According to the conditions of the basic coordinators, we would have to use Reset on this system. However, we can isolate the components involved in the cycle (C, D, E), along with their ancestors (A, B) and compose them into a new component, W. Now between W and X we have an acyclic relationship. So we can use Reset to stabilize E, and one of the acyclic coordinators as appropriate to stabilize E and D. In larger systems this could result in a substantial improvement in

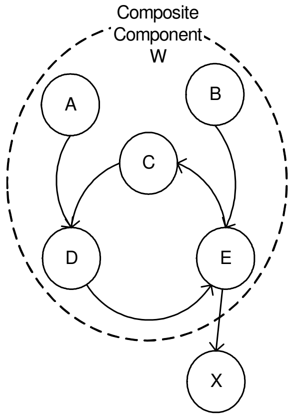


Figure 8: Stabilizing a System with Cyclic Correction

efficiency.

Peer Correction. In [18], a composite component is constructed from two data structures, a heap and a search tree. Ideally, these two structures contain the same information; maintaining the two structures makes different kinds of access efficient. When the composite structure is corrupted, it stabilizes to a state in which both constituents again have the same information. These components are not independent since the link invariant between them is not *true*.

Each constituent acts as parent for the other. When the composite structure is accessed according to the heap, inconsistencies with the search tree are resolved according to the search tree. Similarly, when the composite is accessed according to the search tree, inconsistencies are resolved according to the heap.

Since there is no fixed parent-child relationship, this structure does not fit our setup. But we can accommodate it with nesting by providing custom stabilizers for the composite component.

Systems Without Local Detection and Correction. As mentioned earlier, our setup assumes local detection and correction. If we don't have this property, then we can use nesting to handle it. For example, suppose a component has two parents. Each component contains an integer. The system invariant is that the child component's value is the sum of the values of the parents. The relationships between the child and each parent cannot be given independently. We can incorporate this into our setup by treating all three components as a single composite component and providing appropriate (non-local) stabilizers for the composite.

4 Application in Production Settings

We have developed a toolkit that can be used to perform the coordinator composition. The Dynamic Reconfiguration SubSystem (DRSS, pronounced "Doris") [16, 3], is a .NET framework that allows interceptors to be added to a component. An interceptor can be run based on message traffic between components, or it can be run according to a schedule. An interceptor can call methods on the component, which makes it well-suited for implementing a coordinator. When a coordinator action is begun, it calls on the component's stabilizer methods. If a component does not come with stabilizers, they can be created and, using DRSS, composed with the component.

In presenting our theory of compositional stabilization we have assumed a high degree of atomicity. One component atomically calls another component's method and stabilizers execute atomically. One might argue that these assumptions are impractical, and indeed they are. Fortunately, they are not central to the theory, as we now discuss. This theory was developed in part in response to issues involving sensor networks [12, 5], so it is well-suited to a distributed system with low atomicity.

While we use method calls, other forms of communication can be used. For example, components could communicate via communication registers, shared memory, message passing, shared events as in IO automata, and so forth. Some of these such as message passing imply less atomic communication. This has implications that we discuss next in a general context of refined atomicity.

Detectors. A local detector accesses the component's state and returns a Boolean value indicating whether the component invariant is violated. If the component is distributed, this amounts to distributed predicate detection, a topic that has been well-studied (see, for example, [6]). A similar situation holds for link detectors.

In coordinator guards, several detectors can be invoked atomically: the local detector along with any parent link detectors. These can be run concurrently, with the corresponding command taken when a detector returns a *true* value.

Correctors. Correctors not only access state but also change it. This leads to two issues. First, the component's actions might interfere with the corrector. Second, if the component has parents, those values might change before component is corrected.

To handle the first concern, it might be necessary to run the corrector in a non-synchronous fashion with respect to the component. This would mean inhibiting all actions of the component until the corrector completes.

To handle the second concern, we begin by observing the following. Suppose the system is in a good state and P and Q are neighbors. Let u' and v' be the local states of P and Q, respectively. Suppose P and Q interact with neighbors but not with each other, with reachable state sets U and V, respectively. Then by subsystem preservation, $\forall u, v : u \in U, v \in V : (u \cup v) \models T.Q.P$. Applying this observation, we note that the approach to correction is to form a protected area from the roots so that when a component is corrected, it's parents are continuously good. Suppose a corrector for a component in this situation accesses a good parent's state and begins correcting the component. By the time the corrector has completed, the parent's state may have changed. But if the component is blocked then, as we have observed, the result of the corrector will satisfy the link invariant to the parent.

This discussion assumes that the component is blocked when it is being corrected, as with the Local Blocking and Global Wave coordinators. However, the Wait Free coordinator does not block, and this can cause a problem. Suppose parent P is in good state v' when Q's corrector starts. If parent P interacts with Q after Q's corrector has started but before it has completed, P can change to a good state v that is not normally reachable from v' and hence the final corrector value may not satisfy $T.Q.P$. To prevent this, Q can be blocked while correction is in process.

Sequential Execution. Correction must be sequenced with other actions. In the Local Blocking coordinator we have this command: "(forall R : $L.Q.R$: $b.R(1)$) ; $lc.Q$; $b.Q(0)$". Here we assume that the following things happen in sequence: first, all children are blocked; next, the corrector is run; finally, the component is unblocked. There are two points to consider for distributed components. First, we need to know when each of the individual actions completes so the next one can be initiated. In the distributed setting this might require a termination detector. Second, permitting overlap between the actions could improve efficiency. For example, the blocking on Q could be released before the corrector is complete, provided it is safe to do so.

5 Case Study: The Pursuer Problem in Sensor Networks

We have given the pursuer problem as an example, with intuitive explanations; now we give the example in more detail. We begin by treating the tracker as a centralized component, then discuss its distributed implementation.

We have two components, a pursuer, *Psr*, and a tracker, *Tr*, with a fixed destination, *dest*, over an M by N grid of nodes. The pursuer attempts to reach the destination by means of a shortest path route provided by the tracker. As we saw earlier, the tracker can corrupt the pursuer by giving a bad path and the pursuer can prevent the tracker from forming a good route. The components, including coordinator actions, are given in Fig 9.

```
                Pursuer component, $Psr$
Variables: $ploc, pnext \in M \times N$
Methods: $gpval : return((ploc, pnext))$
Component actions:
      $pnext = ploc \longrightarrow pnext := gtnext(ploc)$   % Get move
[]    $true \longrightarrow ploc := pnext$                   % Make move
Coordinator actions:
      $\neg b.Psr \wedge ld.Psr \longrightarrow b.Psr(1)$         % Detect
[]    $b.Psr \longrightarrow b.Tr(1); lc.Psr;\ b.Psr(0)$          % Correct

                Tracker component, $Tr$
Variables: $troute$ : path over nodes
Methods:
      $gtnext\ (loc)$ : wait until $\neg b.Tr$; $return\ (troute\ (loc))$
Component actions:
      No actions; component is fixed point in invariant.
Coordinator actions:
      $\neg b.Tr \wedge \neg ld.Tr \longrightarrow b.Tr(1)$       % Detect
[]    $b.Tr \longrightarrow lc.Tr;\ b.Tr(0)$                     % Correct
```

Figure 9: Pursuer-Tracker Components

Note that the coordinator actions are added to the component actions;

the original component actions are unchanged. The stabilizers that the coordinator uses are as follows.

- $cd.Psr$: detects $\neg I.Psr$, where $I.Psr$ is $(pnext = ploc) \vee (dist(pnext, dest) < dist(ploc, dest))$.
- $lc.Psr$: sets *pnext* to *ploc*.
- $b.Psr$: Since *Psr* has no parents, no actual blocking takes place. What matters is whether *Psr* is marked for correction.
- $cd.Tr$: Detects $\neg I.Tr$, where $I.Tr$ holds when *troute* is a shortest path and is rooted at *dest*.
- $kd.Tr.Psr$: Detects $\neg T.Tr.Psr$, where $T.Tr.Psr$ holds when *troute* passes through *ploc* and *pnext*, using method *gpval*.
- $lc.Tr$: Sets *troute* as a shortest path from *ploc* to *dest* that includes *pnext*, using method *gpval*.
- $b.Tr$: Besides indicating whether *Tr* is marked for correction, when $b.Tr$ is *true*, attempts by *Psr* to get the next step are stalled until $b.Tr$ becomes *false*.

By our fairness assumptions, if the invariant of *Psr* is violated sufficiently often, it will eventually be detected and *Psr* will be marked for correction. When *Psr* corrects, it first blocks *Tr* so that subsequently *Tr* cannot recorrupt *Psr*. Finally, *Tr* is corrected and all blocking is released.

We have regarded the tracker as a centralized component. It can be realized as a distributed component, as follows. For convenience we assume a reliable underlying communication network. Below we discuss how to include this as a component in the system. The pursuer route is distributed over the nodes. When the pursuer needs the next step, it uses the communication component to get the value at the *ploc* node.

For the coordinator at *Tr*, we let the node containing the destination be the one that runs the coordinator. When the detector is run, it uses the communication component to send a detection wave along the path to check whether the path is shortest and whether it contains *ploc* and *pnext*. In a grid, a shortest path will not contain both up and down links or both left and right links, so there are just four kinds of shortest path (up/left, up/right, etc).

A wave can easily check this. A return wave indicates whether a problem was found; if so, the component is blocked and marked for correction. To block, the coordinator node uses the communication component to broadcast a "block" wave to all nodes so that they inhibit requests from *Psr* for the next step. When the wave completes, the coordinator broadcasts a wave that returns the values of *ploc* and *pnext*; of course, the two waves could be combined. The coordinator uses $lc.Tr$ to calculate the new route and sends it along the route, so each node sets itself appropriately and passes the route to the next one. Finally, the coordinator broadcasts an "unblock" wave to each node.

Now suppose we include the communication component as a component in the system. In the correction relation, *Com* is parent to both *Psr* and *Tr*. In the corruption relation, there is corruption from *Com* to *Psr*: when *Psr* uses *Com* to get the next step, a bad value could cause *Psr* to violate its invariant. We can use nesting to compose *Psr* and *Tr* into a composite component, *PsrTr*. Then there is downward corruption from *Com* to *PsrTr*, and upward corruption within *PsrTr*. The standard component stabilizers can be used to achieve stabilization.

6 Related Work

Existing work on composition of stabilizing components can be classified according to whether components are dependent or independent; whether components can corrupt each other; whether link invariants are used; whether components have meaningful invariants (that is, other than *true*); whether ongoing behavior is permitted in all components; whether convergence is fast; whether it is efficient in good system states; and whether it supports nesting.

Our approach handles both dependent and independent systems, with optional corruption, and with link invariants, meaningful component invariants, ongoing behavior, fast convergence, efficiency during normal system behavior, and nesting.

In [20], components each stabilize independently. Even though they interact, they cannot corrupt each other. There is no notion of a link invariant but there are meaningful component invariants. The system can exhibit ongoing behavior in all components. Convergence is fast since each component can be independently corrected. It is efficient during normal system operation since detectors can be scheduled at any desired rate. There is no notion of nesting.

[9] addresses the issues of modelling and tolerating incorrect software. Components have meaningful invariants. They are independent, so they can be individually corrected without respect to the state of a neighbor. Coordination of stabilization is on a peer basis, so an acyclic structure is not required. Component faults initiate a global recorrection and hence is similar to our GW coordinator, though the coordinator actions are more complex. This approach permits a dynamic relationship between components, in which any component can at any time interact with any other, contrasted with our approach in which the interactions between components are fixed in advance. Components that have been corrected are returned to service before the entire system has stabilized. It is efficient during normal system operation since detectors can be scheduled at any desired rate. There is no notion of nesting.

Traditional stabilization by layers is given by [13, 17], in which higher-level components are oblivious to the existence of lower-level components, and lower-level components can read (but not write) the state of a higher-level component.[1] There is an order in which correction must take place, so this approach is dependent. Components can corrupt each other, but only in a downward way since higher-level components cannot be affected by the state of lower-level ones. The idea of link invariant is not made explicit. Component invariants are usually not trivial. Except for the lowest-level components, all components reach a fixed-point. There is no blocking, so components are always available. It is efficient during normal system operation since detectors can be scheduled at any desired rate. There is no notion of nesting.

"Adaptive programming" [15] is similar to stabilization by layers, except that the components depend on an environment that may change. If the environment achieves a fixed point for a sufficiently long time, the components stabilize with respect to it. There is dependency, but only in the sense of stabilization by layers. Corruption is at most downward. There are meaningful component invariants. Behavior is fixed-point. Convergence is fast. It is efficient during normal system operation since detectors can be scheduled at any desired rate. There is no notion of nesting.

[1] The terms "higher" and "lower" can be confusing. Given a dependency relation that is a dag, we use the term "higher" to mean closer to a root, which stabilizes first. This is opposite to the usage in stabilization by layers where the lower levels stabilize first. To avoid confusion, we consistently use "higher" as closer to the root.

In the “constraint satisfaction” approach [8], correction dependencies with associated link invariants exist between components. System actions do not violate link invariants, so there is no corruption. Component invariants are trivial. There can be on-going behavior. There is no blocking, so components are always available. The hierarchial layering of constraints is a form of nesting.

In “self-stabilization by local checking and correction” [10], corruption is downward and correction is acyclic, as is the case with our Wait Free coordinator. The setup uses link invariants but not meaningful component invariants. Behavior is on-going, convergence is fast, and it is efficient. There is no notion of nesting.

In “recursive restart” [11], the components of a system are organized as a tree based on restartability: if a component in the tree is corrupted, it and all components below it are reset to initial states. Hence the component loses any accumulated history it may have shared with its parent. To prevent corruption cycles, resetting components are blocked until all children are reset, so that no component in a reset subtree gives service until the reset is completed. If the reset begins at the root of the system, the entire system is blocked until the reset is complete. The idea of a link invariant is not made explicit. Components can have meaningful invariants and ongoing behavior The speed of convergence is slow since the entire subsystem must be blocked before any component is released to service. It is efficient during normal system operation since detectors can be scheduled at any desired rate. It supports nesting implicitly since larger components stabilize when constituents do.

In “composite stabilizing data structures” [18], a component is a structure that encapsulates other components. For example, we might want a composition of a heap component and a 2-3 tree component in order to take advantage of the access efficiencies of each. These components have a link invariant between them that specifies that both have the same contents. A wrapper component provides the interface, updates the constituent components, and coordinates stabilization if they happen to have different contents. During stabilization, the constituent components can be mutually evolved by the coordinator until their component invariants and link invariants are satisfied. When the composite components are corrupted, the encapsulating component satisfies the property that (a) eventually both constituents have the same information and (b) new information added is not lost during convergence. There is a dependency between the constituents but no corruption.

The link is explicit and invariants are meaningful. There is ongoing behavior. The constituents give service during stabilization, so we view it as fast. Checking the data structures is done when they are accessed, in an efficient way. Nesting is central to this approach.

7 Conclusions and Future Work

We have given a scalable framework for restoring service to large, distributed systems in the presence of faults that would otherwise render the system useless. The framework, which does not require detailed system knowledge, uses coordinators of varying efficiency depending on the correction and corruption characteristics of the components: when correction and corruption are in the same direction, no blocking is needed; when they are in opposite directions, only local blocking is necessary; otherwise blocking is global. In addition, our system model includes both component and link invariants, thus making it applicable to a wide range of systems.

Future Work. As part of our work with the DARPA NEST program we are planning experiments for large-scale sensor networks involving multiple stabilizing distributed components for routing, time synchronization, and the like. These experiments will demonstrate the utility of our compositional theory in larger systems, and permit us to improve it.

In developing components and stabilizers we would like to work at an abstract level (such as source code) and let a tool (such as a compiler) refine the abstract to an actual concrete system. However, even if an abstract system is stabilizing, a refinement might not be. Refinements only ensure correct behavior from invariant states; from other states, interferences can be introduced that prevent stabilization. Our compositional approach helps mitigate this. If a coordinator causes a refined corrector to be run on a refined component, interferences can occur. However, these are are restricted to the corrector and the component, so a local solution is to stop all component actions and let the corrector run non-concurrently. We will be developing these ideas, to give a theory of compositional refinement.

However, even source code can be hard to deal with: large components can have tens of thousands of lines of code or more. Since specifications are usually simpler than programs, we plan to develop a gray box approach to stabilization. First we identify specifications, using use existing tools such as Bandera [1] to assist as necessary. Next, we develop the stabilizers, based on the specification. Finally, we refine the stabilizers and compose them with the original component.

We have developed DRSS [3], a framework that supports dynamic compositions of components. We plan to use it for our framework by creating a coordinator library so that the coordinator actions can be easily added to an existing system and so be made stabilizing, and we plan to use it for the proposed gray box compositional refinement.

References

[1] The Bandera project. `http://bandera.projects.cis.ksu.edu/`.

[2] The Behave Project. `www.research.microsoft.com/behave/`.

[3] The Dynamic Reconfiguration SubSystem (DRSS). `www.cis.ohio-state.edu/siefast/msr-cont-self-maint/`.

[4] Magellan repository and framework. `research.microsoft.com/reliability/`.

[5] OSU NEST project. `http://nest.netlab.ohio-state.edu/`.

[6] A. Arora and M. Gouda. Closure and convergence: A foundation for fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[7] A. Arora and M. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

[8] A. Arora, M. G. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems. *Journal of High Speed Networks*, 5(3):293–306, 1996.

[9] A. Arora and M. Theimer. On modeling and tolerating incorrect software. Technical Report MSR-TR-2003-27, Microsoft Research, 2003.

[10] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction (extended abstract). In *Proceedings of 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[11] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 125–132, Schloss Elmau, Germany, May 2001.

[12] M. Demirbas, A. Arora, and M. Gouda. A pursuer-evader game for sensor networks. *Sixth Symposium on Self-Stabilizing Systems(SSS'03)*, 2003.

[13] S. Dolev. *Self-Stabilization.* MIT Press, 2000.

[14] M. D. Ernst, J. Cockreil, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[15] M. Gouda and T. Herman. Adaptive programming. *IEEE Transactions on Software Engineering (TSE)*, 17(9):911–921, 1991.

[16] J. Hallstrom, W. Leal, and A. Arora. Scalable evolution of highly available systems. *Transactions of the Institute for Electronics, Information and Communication Engineers (IEICE)*, In Press, 2003.

[17] T. Herman. *Adaptivity Through Distributed Convergence.* PhD thesis, University of Texas at Austin, 1991.

[18] T. Herman and I. Pirwani. A composite stabilizing data structure. In A. K. Datta and T. Herman, editors, *Self-Stabilizing Systems*, LNCS 2194, pages 167–182. Workshop on Self-Stabilizing Systems (WSS), Springer-Verlag, 2001.

[19] W. Leal and A. Arora. Scalable self-stabilization via composition. Technical Report OSU-CISRC-7/03-TR46, Department of Computer Information Science, The Ohio State University, `www.cis.ohio-state.edu`, July 2003.

[20] G. Varghese. Compositional proofs of self-stabilizing protocols. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 80–94. Carleton University Press, 1997.

A Proofs

Given a system S, assume that P, Q and R range over the components of S.

In the proofs we use standard temporal modalities to form temporal predicates. The modalities are $\Box p$ (p always holds), $\Diamond p$ (p eventually holds), $p \mapsto q$ (p leads to q), $\mathsf{X}\ p$ (p holds in the next state) and $p\ \mathsf{U}\ q$ (p until q), with combinations $\Box\Diamond p$ (infinitely often p holds) and $\Diamond\Box p$ (eventually p always holds). Overloading notation, if σ is a computation and r is a temporal predicate then $\sigma \models r$ means that r is true in σ. This is extended in the natural way to sets of computations: $X \models r$ if every computation of X satisfies r.

Given a system with an acyclic relation on the components, the depth of a component wrt the relation is largest number of links from a root to the component.

Theorem 8 (Soundness of Wait Free Coordinator) *Let $\hat{S}$ be a system s.t. the dependency relation, L, is acyclic, and the corruption relation, M, subsets L. Let S be $\hat{S}$ with coordinator WF applied to each component.*

Then true $\mapsto I.S$.

Proof. Define $a.Q$ as the conjunction of component and link invariants on all paths from component Q to roots in L. Define the protected area, $pa.s$ as the state function $\{Q : a.Q\}$. Note that if all components are in the protected area then the system invariant is satisfied, since all component and link invariants hold.

We prove the result by showing safety and liveness properties: a component in the protected area never subsequently leaves it, and eventually all components enter the protected area. Hence eventually all components will be in the protected area simultaneously and so the system invariant will hold.

Let σ be a computation of S.

Safety. Show that the predicate $Q \in pa$ is closed for each Q. That is, $\forall Q :: \sigma \models \Box((Q \in pa)\ \mathsf{X}\ (Q \in pa))$.

Let (s', s) be a subsequence of σ. Let Q be a component and assume $s' \models (Q \in pa)$. Show that $s \models (Q \in pa)$.

- "Detect". The guard is disabled.
- System action.

- Local actions of Q preserve component and link invariants.
- Suppose Q interacts with parent P. Since $s' \models a.Q$, $s' \models (I.P \wedge T.Q.P \wedge I.Q)$, so by the preservation condition, $s \models (Q \in pa)$.
- Suppose Q interacts with child R. By assumption, R does not corrupt Q, so $s \models Q \in pa$.

Liveness. Show that $\sigma \models (pa \neq cmpnt.S \ \wedge \ pa = j) \mapsto (pa \supset j)$, where j ranges over subsets of components of S, where $cmpnt.S$ is the set of components in S and $\supset$ implies inequality.

Let s' be a state in σ s.t. $s' \models (pa \neq cmpnt.S \ \wedge \ pa = j)$. By the acyclic nature of correction, there is Q s.t. $s' \models (Q \notin pa \ \wedge \ (\forall P : L.Q.P : Q \in pa))$. By safety, above, $pa \supseteq j$ holds from s' forward.

After s', either $I.Q$ and the parent link invariants are established spontaneously, or eventually "Detect" will be executed. In either case, eventually there is s s.t. $s \models (I.Q \ \wedge \ (\forall P : L.Q.P : T.Q.P))$. Hence $s \models a.Q$, so $s \models (pa \supset j)$ □

Theorem 9 (Soundness of Local Blocking Coordinator) *Let $\hat{S}$ be a system the following s.t. the dependency relation L is acyclic and one or both of the following hold:*

1. *The corruption relation is upwards ($M^{-1} \subseteq L$).*
2. *The dependency relation, L, is a tree.*

Let S be $\hat{S}$ with coordinator LB applied to each component. Then true $\mapsto$ I.S.

Proof.

Let Q be a component and s a state. Define the state predicate $pt.Q.s$ as *true* if in s there is a tree t rooted at Q s.t. for all R in t (including Q), s satisfies the following.

1. If $P \in t$ then $s \models (I.P \wedge \neg b.P)$
2. If $P, R \in t$ and $L.P.R$ then $s \models T.P.R$.
3. If P is a leaf of t then for all children R of P, $s \models b.R$.

When $pt.Q$ holds in a state, then t forms a protective tree that blocks corruption that might otherwise spread from descendants of Q. The third condition ensures that t is unique.

Let $a.Q$ be the conjunction of component and link invariants from the parents of Q to all the roots in L that are ancestors of Q. If $a.Q \wedge pt.Q$ holds in a state, then Q is protected from above and from below. Further, if $a.Q$ holds continuously and Q is ever blocked, then eventually $pt.Q$ will hold. These observations are summarized in the following lemma.

> **LB Lemma.** Let Q be a component and σ a computation. Assume $(\forall P : L.P.Q : \sigma \models \Box a.Q)$. Then the following hold.
>
> 1. Liveness. $\sigma \models (b.Q \mapsto pt.Q)$.
> 2. Safety. $pt.Q$ is closed in σ.
>
> *Corollary.* Under the theorem assumption, $\sigma |= \Diamond b.Q \Rightarrow \sigma |= \Diamond\Box pt.Q$.
>
> The proof is given later.

To show that $true \mapsto I.S$ it suffices to show $(\forall Q :: \sigma \models \Diamond\Box(I.Q \wedge \neg b.Q \wedge (\forall P : L.P.Q : T.P.Q)))$. The proof is by induction on the depth of Q.

Base case. Depth of $Q = 0$, so Q is a root and has no parents. Suppose the conclusion is false. Then $\sigma \models \Box\Diamond(\neg I.Q \vee b.Q)$. If $\sigma \models \Box\Diamond\neg I.Q$ then by fairness and the "Detect" action, $\sigma \models \Diamond b.Q$. If not, then $\sigma \models \Diamond b.Q$. In either case, by the corollary of the lemma, $\sigma \models \Diamond\Box pt.Q$, and hence a contradiction. So the conclusion holds.

Inductive step. Depth $= n$. Assume the conclusion for components of depth less than n. Let α be a suffix of σ s.t. $(\forall P : \text{depth of } P < n : \sigma |= \Box(I.P \wedge \neg b.P \wedge (\forall W : L.W.P : T.W.P)))$. This implies that for all parents P of Q, $a.P$ holds, so the condition for the lemma is met. As with the base case, assume a contradiction. By the same reasoning, the conclusion holds.

Proof of LB lemma.

Liveness. Assume $b.Q$ holds for some state in σ. Then it holds thereafter until the "Correct" action executes at Q. By fairness, this will eventually happen at some state s_k. Since by assumption all parents of Q are in their invariants at s_k, execution of "Correct" will result in a state s_{k+1} where

$((\forall P : L.P.Q : T.P.Q) \wedge I.Q \wedge \neg b.Q \wedge (\forall R : L.Q.R : b.R))$. This satisfies the definition of $pt.Q$.

Safety. Let (s', s) be a subsequence of σ. Assume $s' \models pt.Q$. Let $t.Q.s'$ be the protected subtree of Q at s'. We consider the following actions.

1. "Detect". The guard is not enabled on any component in $t.Q.s'$. If enabled on any other component, it has no effect on $pt.Q.s'$.

2. "Correct". We consider the following cases.

   (a) Components in $t.Q.s'$. The guard is not enabled.

   (b) Parents of Q. The guard is not enabled since $s' \models a.Q$.

   (c) Parents of components in $t.Q.s'$ other than Q. Let $P \in t.Q.s'$ s.t. $P \neq Q$. Let W be a parent of P. If "Correct" is executed then $s \models b.P$, so $pt.Q$ is still satisfied.

   (d) Children of leaves of $t.Q.s'$. Let P be a leaf of $t.Q.s'$. Let W be s.t. $L.P.W$. We have $s' \models I.P \wedge \neg b.P \wedge b.W$. When "Correct" executes, $lc.P$ establishes $I.P \wedge T.P.W \wedge I.W$. So $s \models (I.P \wedge \neg b.P \wedge T.P.W \wedge I.W \wedge \neg b.W \wedge (\forall X : L.W.X : b.X))$. Hence $s \models pt.Q$.

   (e) Any other component. If "Correct" is executed, it has no effect on $t.Q.s'$.

3. System actions. These actions do not affect blocking. It suffices to show that no component in $t.Q.s'$ is corrupted. We consider the following cases for component $P \in t.Q.s'$.

   (a) W is a parent of P s.t. $W \notin t.Q.s'$. Then L is not a tree. Hence the corruption relation is upwards, and P cannot be corrupted.

   (b) W is a parent of P s.t. $W \in t.Q.s'$. Then by definition of $pt.Q$, $s' \models (I.W \wedge T.W.P)$, so by the preservation condition, P is not corrupted.

   (c) W is a child of P s.t. $s' \models b.W$. Then by the definition of blocking, W does not corrupt P.

   (d) W is a child of P s.t. $s' \models (I.W \wedge \neg b.W \wedge T.P.W)$. Then by the preservation condition, P is not corrupted.

□

Theorem 10 (Soundness of Global Wave Coordinator) *Let $\hat{S}$ be a system s.t. the dependency relation, L, is acyclic. Let S be $\hat{S}$ with coordinator GW applied to each component.*

Then $true \mapsto I.S$.

Proof. We assume the dependency relation is a connected dag; if not, carry out the proof below on each isolated subdag.

Let $\hat{\sigma}$ be a computation of S. Note that action "Fix" reaches a fixed point. Let σ be a suffix of $\hat{\sigma}$ where $\sigma \models \Box(\forall Q :: \mathit{req}.Q \geq \mathit{cor}.Q) \;\wedge\; ((\mathit{req}.Q = \mathit{cor}.Q) \equiv \neg b.Q) \;\wedge\; (\mathit{req}.Q > \mathit{cor}.Q) \equiv b.Q)$.

Let k_0 be the maximum *req* value in s_0 for any component. Let $k_1 = k_0+1$. We have two cases.

- Case 1. Eventually some component assumes a *req* value in σ greater than k_0. That is, $\exists Q : \sigma \models \Diamond(\mathit{req}.Q > k_0)$.
- Case 2. No component assumes a *req* value in σ greater than k_0. That is, $\forall Q : \sigma \models \Diamond\Box(\mathit{req}.Q \leq k_0)$.

The first case is in some sense the "normal" case, in which components are violated but the control variables *req* and *cor* are not. In this case, a violation leads to a global correction wave from the roots. The second case covers the situation where the control variables have been corrupted and a new global wave is not necessary.

Proof of Case 1.

Given state s, define the protected area, $\mathit{pa}.s$ as the following the (possibly empty) rooted subdag of L s.t. s satisfies the following.

- If $Q \in \mathit{pa}.s$ then $I.Q \wedge (\mathit{req}.Q = \mathit{cor}.Q = k_1)$.
- If $P, Q \in \mathit{pa}.s$ s.t. $L.P.Q$ then $T.P.Q$.
- If Q, R are such that Q is a leaf of $\mathit{pa}.s$ and $L.Q.R$ then $\mathit{req}.R = k_1 \wedge \mathit{cor}.R < k_1$.

By the third condition, such a subdag is unique.

We begin by observing that for each root component Q_0 of L, eventually $req.Q_0 = k_1$, as follows. By assumption, eventually some component P will be s.t. $req.P > k_0$. Since this can only happen as a result of "Detect", $req.P = k_1$. "Propagate" will ensure that eventually all components have a *req* value of at least k_1. The *req* value for any component Q cannot exceed k_1 since "Detect" is disabled until $req.Q = cor.Q = k_1$, but this happens only when "Correct" executes at a root. Hence eventually each root Q_0 has $(req.Q_0 = k_1)$.

Note that if all components are in the protected area then the system invariant is satisfied, since all component and link invariants hold, and all components are unblocked.

We prove the result for this case by showing safety and liveness properties: a component in the protected area never subsequently leaves it, and eventually all components enter the protected area. Hence eventually all components will be in the protected area simultaneously and so the system invariant will hold.

Safety for Case 1. Show that the predicate $Q \in pa$ is closed for each Q. That is, $\forall Q :: \sigma \models \Box((Q \in pa)\ \mathsf{X}\ (Q \in pa))$.

Let (s', s) be a subsequence of σ. Let Q be a component and assume $s' \models (Q \in pa)$. Show that $s \models (Q \in pa)$.

- "Detect". $s' \models \neg ld.Q$, so the guard is disabled.
- "Propagate". $s' \models (Q \in pa) \Rightarrow s' \models req.Q = k_1$. The guard is disabled since neighbors all have a *req* value equal to or k_1.
- "Correct".
  - Component Q. $s' \models (req.Q = cor.Q)$, so guard is disabled.
  - Parent P of Q. $s' \models (req.P = cor.P)$, so guard is disabled.
  - Child R of Q. This extends the protected area to include R. Q remains in the area.
- System action. These do not modify the control variables *req* and *cor*, so it suffices to consider corruption.
  - Q and parents. $s' |= (Q \in pa)$, Q and its parents are in their invariants, and the link invariants are satisfied. Hence by the preservation condition, Q is not corrupted.

- Children of Q. If R is in the protected area then Q is not corrupted. If not, then $\mathit{req}.R > \mathit{cor}.R$, and so R is blocked from corrupting Q.

Liveness for Case 1. Show that $\sigma \models (\mathit{pa} \neq \mathit{cmpnt}.S \ \wedge \ \mathit{pa} = j) \mapsto (\mathit{pa} \supset j)$, where j ranges over subsets of components of S and $\supset$ implies inequality.

Eventually $\mathit{req}.Q = k_1$ holds continuously for all Q. This follows from the "Propagate" action as well as the safety property, above. So we consider a suffix of σ where $\mathit{req}.Q = k_1$ holds continuously for all Q.

Let s' be a state s.t. $s' \models (\mathit{pa} \neq \mathit{cmpnt}.S \ \wedge \ \mathit{pa} = j)$. By safety, above, pa never loses components. Hence it suffices to show that there is a component Q s.t. $s' \models (Q \notin \mathit{pa})$ and that there is a subsequent state s s.t. $(Q \in \mathit{pa})$. Proceed as follows.

Let Q be a highest component s.t. $s' \models (Q \notin \mathit{pa})$. Then $\forall P : L.P.Q : s' \models (P \in \mathit{pa})$; else Q is not highest. Then "Correct" is enabled at s'. Eventually either Q enters pa spontaneously, or "Correct" is executed, placing Q in pa. Hence there is a subsequent state s s.t. $(Q \in \mathit{pa})$.

Proof of Case 2. Here we assume that no component assumes a *req* value in σ greater than k_0. There is a suffix α of σ s.t. in all states the *req* value for all components is k_0. This follows from repeated application of "Propagate" plus the fact that *req* values never decrease.

There is a suffix β of α s.t. in all states, $((\mathit{req}.Q = \mathit{cor}.Q) \Rightarrow (I.Q \wedge (\forall P : L.P.Q : T.P.Q))$. To see this, suppose it is not the case for some Q. Then eventually "Detect" will execute and increment $\mathit{req}.Q$ to k_1. But by assumption, $\mathit{req}.Q \leq k_0$, so this is impossible.

Define the protected area $\mathit{pa}.s = \{Q : s \models (\mathit{req}.Q = \mathit{cor}.Q = k_0)\}$. We show both safety and liveness.

Safety for Case 2. Show that the predicate $Q \in \mathit{pa}$ is closed for each Q. That is, $\forall Q :: \beta \models \Box((Q \in \mathit{pa}) \ \mathsf{X} \ (Q \in \mathit{pa}))$. This holds trivially since neither *req* nor *cor* can decrease, and neither can be greater than k_0.

Liveness for Case 2. Show that $\sigma \models (\mathit{pa} \neq \mathit{cmpnt}.S \ \wedge \ \mathit{pa} = j) \mapsto (\mathit{pa} \supset j)$, where j ranges over subsets of components of S. As for Case 1, let s' be a state of β and Q be the highest component in s' that is not in $\mathit{pa}.s'$. Then the parents of Q must be in $\mathit{pa}.s'$. In this case, "Correct" is enabled, and execution of the command will result in a state s s.t. $s \models (\det.Q = \mathit{cor}.Q = k_0)$, so Q is in $\mathit{pa}.s$. □

Contents